BERKELEY LAB

LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

Software Engineering and Process for HPC Scientific Software

Anshu Dubey

With several slides from

Brian Van Straalen
Phil Colella

ATPSEC 2013

Software Process Components

- For All Codes
  - Code Repository
  - Build Process
  - Code Architecture
  - **Coding Standards**
  - Verification Process
  - Maintenance Practices
- If Publicly Distributed code
  - Distribution Policies
  - Contribution Policies
  - Attribution Policies

Coding Standards

- Absolutely essential for code maintainability
  - Consistent code is easier to maintain
  - Someone other than the developer can inspect and make sense out of the code segment
  - Data structures remain more consistent
- Should always include documenting standards also
  - Critical when there is transient population of developers
  - Someone else can understand and maintain your code
  - Easier for users to customize and even contribute code
- Typically involve
  - Naming conventions
  - Inheritance and Code organization

FLASH Coding Standards Namespace

- Namespace directories are capitalized, organizational directories are not
- All API functions of unit start with Unit_ (i.e.Grid_getBlkPtr, Driver_initFlash etc)
- Subunits have composite names that include unit name followed by a capitalized word describing the subunit (i.e. ParticlesMain, ParticlesMapping, GridParticles etc)
- Private unit functions and unit scope variables are named un_routineName (i.e. gr_createDomain, pt_numLocal etc)
- Private functions in subunits other than UnitMain are encouraged to have names like un_suRoutineName, as are the variables in subunit scope data module

Naming Conventions: Within files

- Constants are all uppercase, usually have preprocessor definition, multiple words are separated by an underscore.
  - Permanent constants in "constants.h" or "Unit.h"
    - #define MASTER_PE 0
    - #define CYLINDRICAL 3
  - Generated by setup script in "Flash.h"
    - #define DENS_VAR 1
    - #define NFACE_VARS 6
- Style within routines
  - Variables from Unit_data start with unit_variable: "eos_eintSwitch"
  - Variables begin lowercase, additional words begin with uppercase: "massFraction"

Naming Conventions – How they help

- The significance of capitalizing unit names:
  - A new unit can be added without the need to modify the setup script.
  - If the setup script encounters a top level capitalized directory without an API function to initialize the unit, it issues a warning.
- Variable Style:
  - Immediately clear if variable is CONSTANT, local (massFraction) or global (eos_eintSwitch) in scope

Other Coding Standards

- Implicit none and Use with “ONLY”
  - Purpose is to enforce explicit declaration of every variable
  - If a variable is coming from another module, provide a traceback mechanism
  - Protect ability to give local variable names without worrying about replication and collisions
- Define explicit interfaces for routines
  - Critical for debugging and avoiding seg-faults when for example optional variables are in use
- One externally accessible function per file, function name the same as file name
- Documentation standards include API description and examples for the use of the function

Software Process Components

- For All Codes
  - Code Repository
  - Build Process
  - Code Architecture
  - Coding Standards
  - **Verification Process**
  - Maintenance Practices
- If Publicly Distributed code
  - Distribution Policies
  - Contribution Policies
  - Attribution Policies

Verification

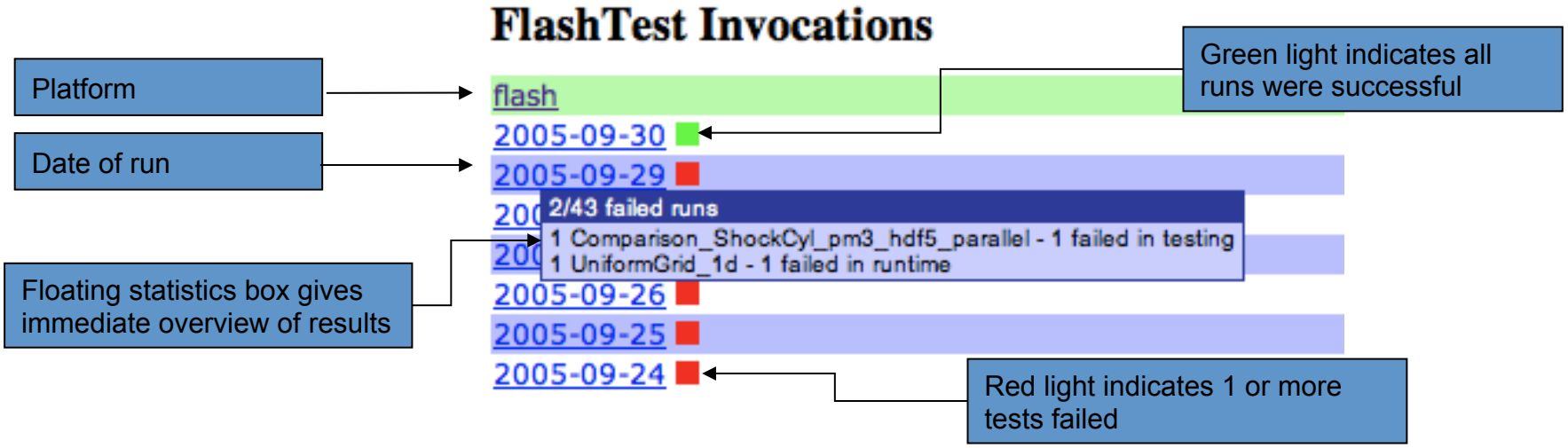
- Codes obviously need to be verified for correctness
- There is no such thing as a bug-free code
- A code is only as robust as the most rigorous test designed for it
- Devising a good test is at least as important as a good algorithm design
- Multi-component code testing needs
  - Unit test to verify a single functionality
    - May need to be done in more than one way
  - Other tests that combine components in many different ways
  - Combinations increase non-linearly with code components

What makes a good test

- Exercises the target code component such that all possible execution paths are explored (nearly impossible to achieve)
- Has minimal dependency on other code components not being tested with this test
- Has alternative way (analytical or semi-analytical, or through a completely different set of operations) of arriving at the same solution if a unit test
- Gives reproducible results if a regression test
- Does not take very long to run
- Produces easy to verify results

The Test Suite

- Runs a variety of problems on multiple platforms on a daily basis
- A platform is defined as a combination of hardware, OS and compiler suite
- In-house software manages automated runs
- Also provides web interface for inspection and modification of tests



FlashTest

Selection of Tests : The Matrix

	UHD	Self Gravity	Particles
Uniform Grid	4		1
AMR	5	2 3	
HDF5 IO	3	3	
PnetDCF IO	4 5		5

- 1 ./setup unitTest/Particles –auto
- 2 ./setup unitTest/Gravity/PoisTest –auto
- 3 ./setup Jeans –auto +uhd +parallelIO +pm4
- 4 ./setup Sedov –auto +pnetcdf +ug
- 5 ./setup Pancake –auto +pm4 +pnetcdf

Selection of Tests : The Methodology

- All unit tests
- Setups corresponding to ongoing research at the center
- Tests known to be sensitive to perturbations
- Tests known to exercise solvers in unusual ways
- Least complex setups to cover the empty spots

What makes a good test-suite

- Verifies the code in every possible meaningful configuration (again impossible to achieve)
- In the absence of comprehensive coverage provides a wide coverage with available resources
- Verifies the code on all supported hardware and software stack
- Is able to report on detected errors in easy to interpret ways
- **Runs regularly and catches bugs introduced into the code base as early as possible**

FLASH Example: The Tests Collection

test type	approach	coverage	examples	done by
unit test	use alternative way to generate verification data	a capability or a solver	guard cells, particle integration	test-suite software
comparison test	against appro-ved benchmark	interoperability among units and apps	advection, shock tube, rotor	test-suite software
restart test	against two approved benchmarks	transparent restart	advection shock tube rotor	test-suite software
target platform	manual verification	specific application	RTflame	human experts
benchmark update	manual verification	affected tests	solver upgrade	human experts
populating new test platform	combination of manual and automated	all tests	compiler upgrade	human experts and test suite software

Software Process Components

- For All Codes
  - Code Repository
  - Build Process
  - Code Architecture
  - Coding Standards
  - Verification Process
  - **Maintenance Practices**
- If Publicly Distributed code
  - Distribution Policies
  - Contribution Policies
  - Attribution Policies

Maintenance Practices

- Repository management
  - Should you have a gatekeeper
  - How far do you allow the branches to diverge
  - How much access control do you apply
- Verification management
  - Monitoring the regression tests
  - Prioritization of efforts : how long do you let a failing test go on failing
- Coding Standards management
  - How do you verify that the new code adheres to coding and documentation standards
- Documentation
  - What fraction of developer time reasonable

Maintenance Practices : FLASH Example

- Repository management
  - No designated gatekeeper
  - The collaborative development branch is the only one where outsiders have access
    - It is also the primary development branch
  - Development branches do exist for specific projects
    - INS, core-collapse for example
  - The code group maintains a weekly merge schedule
  - The concerned developers are expected to make sure that the issues flagged by the merges are resolved before the next project is scheduled to merge

Maintenance Practices : FLASH Example

- Repository management
  - For every development branch if there is a production schedule there is a corresponding **production** branch
  - Stable revisions of the development branches are tagged and periodically merged to production branch
  - Campaigns branch off from the production branch
    - No forward merges occur on these branches
    - Backward merges are rare, but they do happen
    - Usually very limited manual merges of individual files or directories
- It all works only if all participants buy into the practice
- Typical pitfall : someone not checking in their work regularly, their working copy diverges from the repo, updates become a headache

Verification Management

- All developers are expected to provide tests for new capabilities added to the code
- The tests get added to the test-suite
- All developers are expected to monitor the test-suite and resolve the failing tests in a timely manner
- Usually someone from the group takes on the responsibility of monitoring the overall health of the test-suite
- We have gone to a great deal of trouble to automate many of the test-suite functions
- **The test-suite is taken very seriously at FLASH, and all those who have gone on to other places and still use FLASH, start their own versions.**

Coding Standard Management : FLASH Example

- Code is F90 based, compilers tend to be very tolerant of bad code
- Extremely easy to let non-maintainable code proliferate
  - Example : you can violate variable scoping by simply putting in the “use” anywhere, it is valid F90 code
  - Function prototypes (interfaces in F90) are not necessary, you can eat arguments and not find out until it has become hard to debug because it is so old
- Set of scripts that run nightly and flag the violations in coding and document standards
- Periodically (most often just before releases) those violations get resolved

Documentation : How much

- A well maintainable code is likely to have 25-30% of its source as inline documentation
  - More is even better
  - Not doing that is the surest way of a code component to become unsupported (and eventually disappear from the code base) once its developer has moved on
  - Even otherwise, in a common code it is a requirement that others can read and make sense out of your code
  - You might forget why you did what you did
- The APIs should be really well documented in terms of their function, inputs and outputs, the correct range of values for inputs and expected outcome for those values.
  - Examples of use are even better

Documentation : How much

- If the code is public, other type of documentation becomes necessary
  - User's guide
  - Online resources
  - FAQ's or equivalent
- If the code accepts contributions from external users then even more documentation becomes necessary
  - Published coding standards
  - Coding examples
  - Developer's guide

  FLASH Example

Software Process Components

- For All Codes
  - Code Repository
  - Build Process
  - Code Architecture
  - Coding Standards
  - Verification Process
  - Maintenance Practices
- **If Publicly Distributed code**
  - **Distribution Policies**
  - **Contribution Policies**
  - **Attribution Policies**

Variety of User Expertise

- Novice users – execute one of included applications
  - change only the runtime parameters
- Most users – generate new problems, analyze
  - Generate new Simulations with initial conditions, parameters
  - Write alternate API routines for specialized output
- Advanced users – Customize existing routines
  - Add small amounts of new code where their application resides
- Expert – new research
  - Completely new algorithms and/or capabilities
  - Can contribute to core functionality

Distribution Policies

- The licensing agreement
- Distribution control
- What is included in the release
- How often to release

FLASH Example

- A custom licensing agreement
- Source code is included, can be modified, but cannot be redistributed
- More than 3/4 of the usable code base is distributed
- Once or twice a year full releases, patches in-between

Contribution Policies

- Balancing contributors and code distribution needs
  - Contributors want their code to become integrated with the code so it is maintained, but may not want it released immediately
    - Not exercised enough
    - Contributor may want some IP protection
- Maintainable code requirements
  - The minimum set needed from the contributor
    - Source code, build scripts, tests, documentation
- Agreement on user support
  - Contributor or the distributor
- Add-ons not included with the distribution, but work with the code

Contribution and Attribution Policies: FLASH Example

- Code accepted with the understanding that it will eventually be distributed
- Pre-negotiated period of time when the code exists in FLASH repo but is not released
- The contributor provides user support also for negotiated time (usually that doesn't stop)
- The contribution does need to include the makefile snippet and appropriate tests that can be included in the test suite
- At least one example setup for users and its appropriate documentation is needed if it is a new capability
- If it is an alternative implementation of a new capability then the documentation only for the code is sufficient
- All contributions are acknowledged in user's guide and release notes. The contributors can also provide publications to be cited if their code is used